Technical Report
CMU/SEI-95-TR-002
ESC-TR-95-002

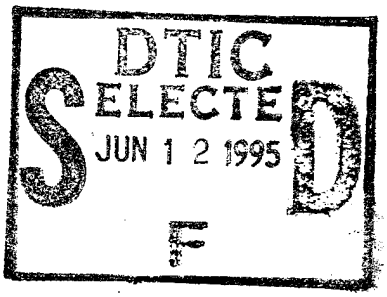
Carnegie-Mellon University
Software Engineering Institute

Object-Oriented Software Measures

Clark Archer

Michael Stinson

April 1995



Technical Report
CMU/SEI-95-TR-002
ESC-TR-95-002
April 1995

Object-Oriented Software Measures

Clark Archer
Winthrop University

Michael Stinson
Central Michigan University

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

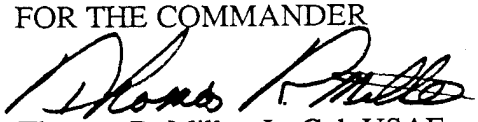
SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994.

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Tables

List of Figures

Object-Oriented Software Measures

Abstract: This paper provides an overview of the merging of a paradigm and a process, the object-oriented paradigm and the software product measurement process. A taxonomy of object-oriented software measures is created, and existing object-oriented software measures are enumerated, evaluated, and placed in taxa. This report includes an extensive bibliography of the current object-oriented measures that apply to the design and implementation phases of a software project. Examples of computation of representative measures are included.

1. Introduction

It is quite clear that measurement is necessary for the software development process to be successful. In addition, the path to controlling and improving the software design process may lie in the use of an object-oriented design approach. The recent movement toward object-oriented technology must also include the processes that control object-oriented development, namely software measures. Tom DeMarco summarizes the essence of these sentiments by stating, "You cannot control what you cannot measure" [DeMarco 87]. Measurement encompasses many aspects of the software life cycle. The emphasis of this document is on the design and implementation phases of an object-oriented approach. This report investigates the software product measures that exist in these phases and develops a taxonomy for these measures.

1.1. Reasonable Characteristics of a Measure

A measure is a numerical value computed from a collection of data. Before examining the details of software measures (often called metrics), let's consider which properties of a measure, in general, that are reasonable. A measure should have the following characteristics to be of value to us:

- **The measure should be robust.** The calculation of the measure is repeatable and the result is insensitive to minor changes in environment, tool, or observer. The measure is precise, and the process of collecting the data for the measure is objective.

- **The measure should suggest a norm, scale, and bounds.** There is a scale upon which we can make a comparison of two measures of the same type, and so conclude which of the two measures is more desirable. For example, there is a realistic lower bound, such as zero for number of errors.

- **The measure should be meaningful.** The measure relates to the product, and there should be a rationale for collecting data for the measure.

Often, one measure alone is insufficient to measure the features of the design paradigm or to accomplish the objectives of the software project. This suggests that a collection or suite of measures is needed to provide the range and diversity necessary to achieve the software project's objectives. A suite of measures adds an additional consideration.

- **A suite of measures should be consistent.** If a smaller value is better for one type of measure in the suite, then smaller is better for all other types of measures in the suite.

In addition, the data gathering process that produced the data from which a measure is computed should be carefully orchestrated. Data gathering without a reason is not likely to produce meaningful results. Although data acquisition is not the topic of this report, data gathering issues should not be ignored. Fenton [Fenton 91] covers this topic quite well.

1.2. Overview of Software Metrics

Many people are reluctant to use the term metric in reference to software. *The American Heritage Dictionary* (Mifflin, 1991) defines a metric as:

1. designating, pertaining to the metric system, or

2. a standard of measurement.

Mathematicians define a metric more rigorously; they use the term to apply to a real-valued set function that measures the distance (as defined by the metric) between two objects in the set. In his text on topology, Mansfield [Mansfield 63] defines a metric as follows:

> Let A be a set of objects, let R be the set of real numbers, and let ρ be a one-to-one function such that $\rho: A \otimes A \rightarrow R$, where $\otimes$ denotes the Cartesian product of A with A. Then, ρ is a metric for A if and only if
>
> - $\rho(\alpha,\beta) \geq 0 \quad \forall \alpha,\beta \in A$,
> - $\rho(\alpha,\beta) = 0 \Leftrightarrow \alpha=\beta$,
> - $\rho(\alpha,\beta) = \rho(\beta,\alpha) \quad \forall \alpha,\beta \in A$, and
> - $\rho(\alpha,\gamma) \leq \rho(\alpha,\beta) + \rho(\beta,\gamma) \quad \forall \alpha,\beta,\gamma \in A$.

For the purposes of this document, the term software metrics will mean measurements made on a software artifact. There are two important components of the software artifact that are measured for our purposes: the artifact's design specification document and its coded implementation. The early computed metrics of Halstead, McCabe, and Albrecht, introduced in the late 1970s, were usually based on only the coded end-product. These are the software science metrics [Halstead 77], cyclomatic complexity metric [McCabe 76], and function point analysis [Albrecht 79] that prevailed in the early 1980s as software product measures. These earlier approaches to measuring software artifacts were based on the software specification methods of the time and the programming languages that supported these methods. The paradigm that dominated software development in the late 1970s and early 1980s was a top-down structured development.

The need to obtain measures of the intended software product early in the life cycle grew along with the need to estimate the cost of the product. Since software production is a labor-intensive process, and labor is based on time spent, the cost of the software product is directly related to certain of its features such as the size of the product, the complexity of the product, and expected reliability of the product. Any metric that could be computed at the design stage and could measure any of these features is useful for predicting the product's cost.

1.3. Software Measure

The concept of a metric measuring of the *distance* between two objects in a set A has very little meaning in the world of software. Why would we want to measure the distance between two software products or two software specifications? It does, however, make sense to measure software product X and software product Y, and then, to compare the two measures. We also note that there is no standard of measurement for software artifacts that is universally accepted. Based on both the dictionary and mathematical definitions of metric, we see that the term *software metric* is not appropriate. The preferred term is *software measure.*

1.3.1. Weyuker's Measure Properties

Many issues arise as to what constitutes, and what are the acceptable properties of, a software measure. Elaine Weyuker has brought together nine properties that a software product measure should have [Weyuker 88]. Many authors have used these properties as a standard against which to evaluate their own measures.

> "All the measures considered depend only on the syntactic features of the program" [Weyuker 88].
>
> Let P, Q, and R be programs.
>
> $P \equiv Q$ means that P and Q halt on the same input.

P;Q means that P is augmented by Q. (An appending of Q to P)

The measure of P is denoted by | P |.

Nine properties of measures:

1. $(\exists P)(\exists Q)\ (\ |P| \neq |Q|\ )$.
2. Let c be a nonnegative number. Then there are only finitely many programs of measure c.
3. There are distinct programs P and Q such that $|P| = |Q|$.
4. $(\exists P)(\exists Q)\ (P \equiv Q$ and $|P| \neq |Q|\ )$.
5. $(\forall P)(\forall Q)\ (\ |P| \leq |P;Q|\ )$ and $(\ |Q| \leq |P;Q|\ )$.
6. $(\exists P)(\exists Q)(\exists R)\ (\ |P| = |Q|\ ) \& (\ |P;R| \neq |Q;R|\ )$ and

   $(\exists P)(\exists Q)(\exists R)\ (\ |P| = |Q|\ ) \& (\ |R;P| \neq |R;Q|\ )$.
7. There are program bodies P and Q such that Q is formed by permuting the order of the statements of P; and $|P| \neq |Q|$.
8. If P is a renaming of Q, then $|P| = |Q|$.
9. $(\exists P)(\exists Q)\ (\ |P| + |Q| < |P;Q|\ )$.

1.3.2. Comments on Weyuker's Properties

Property number one [$(\exists P)(\exists Q)\ (\ |P| \neq |Q|\ )$] reflects the idea that a measure that assigns all programs the same value is not a measure. Property number two (for a nonnegative number c there are only finitely many programs of measure c) is the non-coarseness property: it places a constraint on property one by stating that only a finite number of programs can be assigned the same measure. Property number three (there are distinct programs P and Q such that $|P| = |Q|$) is often called the non-uniqueness property: two different products can have the same measure value. Property number four [$(\exists P)(\exists Q)\ (P \equiv Q$ and $|P| \neq |Q|\ )$] states that two software products can possess the same functionality but not have equal measure values. Property number five [$(\forall P)(\forall Q)\ (\ |P| \leq |P;Q|\ )$ and $(\ |Q| \leq |P;Q|\ )$] is a monotonicity requirement: a combination (concatenation) of two products can never have a measure value less than either of the products taken individually. Property number six [$(\exists P)(\exists Q)(\exists R)\ (\ |P| = |Q|\ ) \& (\ |P;R| \neq |Q;R|\ )$ and $(\exists P)(\exists Q)(\exists R)\ (\ |P| = |Q|\ ) \& (\ |R;P| \neq |R;Q|\ )$] states that there exist products whose measure values are the same, but the augmentation of either product by a third product can produce measure values that are not the same. Property number seven [there are program bodies P and Q such that Q is formed by permuting the order of the statements of P; and $|P| \neq |Q|$] states that there are software products whose measure value can be affected by a permutation in the order of program statements. Property number eight (if P is a renaming of Q, then $|P| = |Q|$) is the "carbon copy" property

indicating that the measure value is not affected by any isomorphic transformation of the original product. Property number nine [$(\exists P)(\exists Q)$ ($|P| + |Q| < |P;Q|$)] is the most controversial of the nine properties. This property states that augmentation increases the measure value for some software products.

Weyuker's properties are concerned with computer programs. What features of computer programs do these properties encompass? The answer to this question is unclear. Consider property number five which states "for all programs P and Q the measure of program P augmented by program Q is greater than or equal to both programs P and Q alone." This property is reasonable if the feature of concern is program size and the measure is the number of lines of executable source code. However, for the same feature program size and the same measure number of lines of executable source code, property number five is in conflict with property number six. Property six states, "there exist programs P, Q, and R such that programs P and Q can have the same measure and the measure of P augmented by R is different from Q augmented by R." This property is not true for lines of code that are used as the measure and, in fact, is not true for most size measures, suggesting that Weyuker's properties encompass some feature other than program size.

Since the title of Weyuker's article is "Evaluating Software Complexity Measures," the properties must also involve complexity. McCabe [McCabe 76] introduced a measure called the cyclomatic complexity metric $v = \pi + 1$, where π is the number of predicates in a program. A predicate in a program is a Boolean expression having one of the forms:

$$B1 = B2,\ B1 \neq B2,\ B1 < B2,\ B1 > B2,\ B1 \leq B2,\ \text{or}\ B1 \geq B2,$$

where B1 is an identifier and B2 is either a constant or an identifier. To use the predicate count approach to compute McCabe's metric, all statements involving compound Boolean expressions are reduced to a sequence of statements with only predicates in them. Careful calculation indicates that Weyuker's property five is satisfied and property six is not satisfied. Thus, Weyuker's properties do not encompass McCabe's view of complexity.

Halstead, however, introduced a measure that does satisfy property six. The measure (called an effort measure) measures the effort involved in producing an algorithm [Halstead 77], but the measure is difficult to compute; it involves the counts of the total occurrence of operators and operands and the counts of unique operators used and unique operands. Halstead's effort measure is implementation-dependent. Furthermore, Weyuker proves algebraically that the Halstead effort measure does not satisfy her property number five, but does satisfy her property number six.

Which features, then, of software products are encompassed by Weyuker's properties? Fenton resolves this issue by stating, "Properties five and six are relevant for very different (and incompatible) views of complexity. Hence it is impossible to define a set of axioms for a completely general view of 'complexity'" [Fenton 91]. This suggests that software products have features that can be identified and grouped into categories that include features, measures, and axioms for these measures.

Weyuker's set of properties is a seminal effort in establishing a basis for evaluating software measures. Some of the properties should apply to all software measures; some apply to a chosen few features that we may wish to measure. Property number two, for example, is a property that all measures should satisfy. Simply stated, this property requires that a measure not be "too coarse." Yet, property number two is not satisfied by McCabe's cyclomatic complexity measure, in which too many programs would be assigned the same measure. We provide an example of this in the computation of the example measures in sections 7.2.1 and 7.3.1 of this report.

That software products have features that have conflicting properties is evidenced by established and accepted measures that do not satisfy some set of Weyuker's properties. Once a design and implementation paradigm is chosen, the features of concern of the software products to be produced should be isolated and grouped into categories. Measures can be selected for each category, and lists of properties can be developed for these measures. Weyuker's properties can be used as a basis for selecting these properties. This also suggests that a collection of measures may be appropriate for the application as opposed to a single measure.

1.4. Emergence of a New Paradigm

A new paradigm became popular in the mid 1980s that began to affect the way software developers viewed software analysis and design. This paradigm, the object-oriented paradigm, added a new level of complexity to the study of software measures. Are software products produced under object-oriented techniques measurable by existing software measures, or does a new body of measures need to be invented? What is the current state of the discipline relative to *object-oriented measures*?

2. Overview of the Object-Oriented Approach

One of the problems of any software project is simply to be able to manage the concepts, flow of control, and data. Many software engineers feel that part of the solution to this problem lies in the use of an object-oriented approach [Booch 94], which groups both data and procedures (called methods) into an entity called an *object*. These objects allow a programmer, designer, and analyst to examine larger cognitive blocks of a project, and thus clarify the programming process. While these chunks may clarify the process of software design, they raise some questions as to the origin of the objects, the structures under which they reside, and whether and how we can examine objects for complexity.

2.1. Origins of the Paradigm

Ole-Johan Dahl and Kristen Nygaard of Norway created the seminal work on an object-oriented language with their introduction of Simula67 in 1967. As the name implies, Simula67 was generally used for simulation modeling and proved to be a significant influence on later object-oriented languages. Smalltalk, developed at XEROX in Palo Alto in the 1970s, was the next major development of an object-oriented language. Smalltalk was followed by a number of languages that either were object-oriented from inception, such as Eiffel, or revamped a previous language to include object-oriented capabilities, such as C++, Object Pascal, and Ada95.

2.2. Elements of the Object-Oriented Approach

The basic element in an object-oriented system is an *object*. An object is an encapsulation of both data and functionality with the added support of message passing and inheritance. We refer to the data in an object as the attributes of the object, while the functionality is provided by the methods. These two entities—attributes and functionality—form a single logical entity, an object. This contrasts with the more traditional structured programming, which considers data separately from the procedures that act on them. Such a logical grouping of data, along with the procedures that will affect them, gives a conceptual as well as a physical basis for an object. For example, the same logical grouping mimics some nice properties that humans take for granted in the way we conceptualize everyday objects. The old joke about whether the price of the car includes a steering wheel indicates to us that we have predetermined that the object has the functionality of steering embedded in the concept of the car (object).

Objects themselves are usually created through an instantiation process that uses a general template called a *class*. The template contains the characteristics of the class, without containing the specific data that needs to be inserted into the template to form the object. This lack of specification is analogous to the well-known concept of referencing a stack without specifying what is in the stack. That is, certain stack features are well known and understood, although we do not yet know the type of elements in the stack.

Classes are the basis for most design work in objects. Although the purpose may be to instantiate objects, that particular task can be delayed as the higher level of abstraction is created. Classes are either superclasses (root classes), created with a set of basic attributes and methods, or subclasses, meaning they inherit the characteristics of the parent's class and may add (or remove) functionality as desired.

A superclass may be created with general characteristics found in all classes of a particular relationship. For example, a superclass car might contain the characteristics found in most cars: steering, braking, and power for locomotion. Subclasses of the superclass car would inherit all of these characteristics simply because they are a descendant of the superclass. From the perspective of the class that inherits the characteristics, the inheritance forms an IS_A relationship. This type of relationship forms what we call a *class hierarchy lattice*.

Another type of relationship, interaction between classes, may take two different forms: classes may share data through aggregate or component grouping, or classes may share objects.

Aggregate classes interact through messages, which are directed requests for services from one class (called a client) to another class (called a server). Notice that the class that makes the request depends upon the collaborating server class; the client is said to be *coupled to* the server. The serving class may have no dependence on the class using the requested material, so clearly this relationship is not commutative. The relationship in which two or more different classes form a component, thus developing a PART_OF, is also called a HAS_A relationship.

If one object *uses* another object through another class, the dependency is now upon the attributes and methods of the used object. Because of this additional complexity, we choose to consider the *uses* relationship separately from simply the passing of attributes.

3. Terminology

3.1 Terms Specific to Object-Oriented Analysis and Design

In this report, we treat the term *object* as a primitive term. Objects have attributes, methods, and identity (a name). The following terminology is a partial adaptation of Booch's set of terms [Booch 94]. We provide these definitions so that the terminology used to describe object-oriented software products is as uniform as possible.

Abstraction. The essential characteristics of an object that distinguish it from all other kinds of objects, and thus provide, from the viewer's perspective, crisply-defined conceptual boundaries; the process of focusing upon the essential characteristics of an object.

Aggregate object (aggregation). An object composed of two or more other objects. An object that is *part of* two or more other objects.

Attribute. A variable or parameter that is encapsulated into an object.

Class. A set of objects that share a common structure and behavior manifested by a set of methods; the set serves as a template from which objects can be created.

Class structure. A graph whose vertices represent classes and whose arcs represent relationships among the classes.

Cohesion. The degree to which the methods within a class are related to one another.[1]

Collaborating classes. If a class sends a message to another class, the classes are said to be collaborating.

Coupling. Object X is coupled to object Y if and only if X sends a message to Y.

Encapsulation. The process of bundling together the elements of an abstraction that constitute its structure and behavior.

Information hiding. The process of hiding the structure of an object and the implementation details of its methods. An object has a public interface and a private representation; these two elements are kept distinct.

Inheritance. A relationship among classes, wherein one class shares the structure or methods defined in one other class (for single inheritance) or in more than one other class (for multiple inheritance).

[1] Here, cohesion is limited to *cohesion within a class.*

Instance. An object with specific structure, specific methods, and an identity.

Instantiation. The process of filling in the template of a class to produce a class from which one can create instances.

Message. A request made of one object to another, to perform an operation.

Method. An operation upon an object, defined as part of the declaration of a class.

Polymorphism. The ability of two or more objects to interpret a message differently at execution, depending upon the superclass of the calling object.

Superclass. The class from which another subclass inherits its attributes and methods.

Uses. If object X is coupled to object Y and object Y is coupled to object Z, then object X uses object Z.

3.2. Comments on Terminology

In other contexts, cohesion can apply to many other aspects of the object-oriented paradigm: cohesion of classes within a superclass, cohesion of instances of objects for a specific class, or cohesion of superclasses within a system. We apply the term cohesion to methods within a class. Some authors have considered various types of cohesion in the context of structured analysis, such as temporal, functional, and logical cohesion. Researchers may choose to explore these types of cohesion in the context of object-oriented analysis and design.

4. The Nature of Object-Oriented Software

4.1. Different Features Of Object-Oriented Products

One feature that makes objected-oriented software products different from earlier or conventional software products is the use of procedures and subprograms. By the mid-1960s, subprograms were used as a means of abstracting the main functions of a larger software artifact. The realization that subprograms could serve as an abstraction mechanism had three important consequences. First, languages were invented that supported a variety of parameter-passing mechanisms. Second, the foundations of structured programming were laid, manifesting themselves in language support for the nesting of subprograms, and for the scope and visibility of declarations. Third, structured design methods emerged, offering guidance to designers using subprograms as the basic building blocks [Booch 94] for large systems.

The need to design and program larger applications caused several refinements to structured design methods. Among these were the structured analysis and design technique (SADT), Jackson structured programming, and the programming concept of separate compilation. In retrospect, software in the 1970s and early 1980s saw procedure-oriented programming, poor support for data abstraction, lack of strong data typing, and global use of blocks of data.

Today, the conventional technique of structured programming is still procedure-oriented, but is supported by programming languages that support separate compilation of modules, data abstraction, strong data typing, and data encapsulation. That structured programming is still procedure-oriented indicates an early emphasis on implementation in the life cycle. Today and in the past, a major portion of the life cycle is spent on implementing the design.

In contrast, object-oriented programming places greater emphasis on the design phase of the software life cycle. The essence of the object-oriented design is that it decomposes the system into objects, the basic building block of the object-oriented approach; gathers together the data and the functions to be performed on the data; and encapsulates the data and functions (methods) within the object.

The feature that makes object-oriented software products different from the conventional procedure-oriented software products is the object itself. The features of the object that become measurable are the number of attributes the object contains, the number of methods the object has, the number of methods called from other objects, the number of methods outside the called object, and the placement of the object in the class hierarchy structure.

Unique features of object-oriented programming and design impose added complexities on the measuring process. These features—message passing, inheritance, and polymorphism—require a suite of measures designed to handle them.

4.2 Shared Features Of Object-Oriented Products

Abstract data types exist in conventional procedure-oriented programming languages, and classes can be implemented as abstract data types in most of the existing object-oriented languages. However, one of the key differences between the conventional implementation and the object-oriented implementation is the concept of inheritance. Without inheritance, every class implemented as an abstract data type would be an isolated unit.

The methods of an object are similar to the functions, programs, or subprograms that are used in conventional programming, except that their functionality is limited to specific object classes. An object's methods are measurable. Each of these coded software modules can be measured by the earlier, more conventional measures. Examples of such measures are Halstead's software science metrics, lines of code, McCabe's cyclomatic complexity metric, and Albrecht's function points.

5. A Taxonomy for Object-Oriented Software Measures

The object-oriented design approach gives rise to a natural taxonomy that incorporates the salient features and properties of an object-oriented system. Our taxonomy captures these properties hierarchically. It begins with the high-level characteristics of an object-oriented system and moves down to the low-level characteristics.

System. We place the system and its components at the highest level. Although a system can be subdivided into components, we view the components as acting as a system. Also, the characteristics of a good component are those of a good system and vice versa. The measurable characteristics of a system might include the number of classes or class lattices in the system.

Coupling and Uses. Classes often interact with other classes to form a subsystem. Characteristics of this interaction may indicate a complexity resulting from too much coupling, or from using objects derived from objects that have been obtained from yet another object. Such complexity can complicate the programming process. Uses and coupling are related issues; uses is defined in terms of coupling. We feel that the origin of uses and coupling in the interaction of classes makes them a single taxon: both capture the interaction of classes.

Inheritance. Classes are found in a class structure diagram, often called a class hierarchy lattice. Visible in the lattice are the inheritance relationships between classes and their parents—the properties shared by both. Such relationships may indicate to a designer where changes would improve the development. The lattice itself contains interesting characteristics, such as the depth and breadth of the lattice.

Class. Next on the taxonomy is the class, which contains the methods. The class may have methods that are unnecessary or too complex to work together. It may have extraneous data that complicates the programming process. The class may be linked too closely to other classes or have characteristics that make it an excellent candidate for inclusion in a library.

Method. Attributes and methods occur at the finest level of detail. While the attributes and data structures are fairly well understood, the methods are usually developed much like procedures are in structured programming. The characteristics of procedures are known, and techniques to analyze them are common. Methods have the additional complexity of calling objects other than the object that contains them.

Careful inspection shows that this taxonomy encompasses all the characteristics of object-oriented software products and captures the features of the design at the appropriate levels. These taxa also give the best insight into potential areas of concern, such as depth of inheritance, cohesion, size of objects, and system structure. (See Table 5-1.)

5.1. Examples of Measures and Their Corresponding Taxon

Table 5-1: Measures and Their Corresponding Taxon

	System	Coupling & Uses	Inheritance	Class	Method
Abreu 93	SC1 SR1 SR2 SR3			CC2 CR1 CC3 CR2 CR3	
Banker 91	OC OP RL			RFC	
Chen 93		CCM OCM	CHM	OXM RM OACM ACM CM	
Chidamber 94		CBO	DIT NOC	WMC RFC LOCM	
Coppick 92					SSM MCC (Flavors)
Laranjeira 90	Size				
Lee 93	HC PC			CC	MC
Li 93		MPC		DAC NOM Size2	Size1 (ClassicAda)
Moreau 90ab		GSDM	IL		SSM MCC (C, C++)
Sharble 93		NOT VOD		WAC	
Tegarden 92					SSM MCC LOC
Williams 93	CBC CSC	CCR COU			
Total Measures	12	10	4	19	9

5.2. Explanation of Table Acronyms

Abreu 94	SC1 - System Complexity (total length of inheritance chain)
	CC2 - Class Complexity (progeny count)
	CC3 - Class Complexity (parent count)
	CR1 - Class Reuse (% of inherited methods that are overloaded)
	CR2 - Class Reuse (number of times class is reused "as is")
	CR3 - Class Reuse (number of times class is reused with adaptation)
	SR1 - System Reuse (% reused "as is" classes)
	SR2 - System Reuse (% reused classes with adaptation)
	SR3 - System Reuse (library quality factor)
Banker 91	RFC - Raw Function Counts
	OC - Object Counts (count of classes)
	OP - Object Points
	RL - Reuse Leverage
Chen 93	OXM - Operation Complexity Metric (within a class)
	OACM - Operation Argument Complexity Metric
	ACM - Attribute Complexity Metric
	OCM - Operation Coupling Metric
	CCM - Class Coupling Metric
	CM - Cohesion Metric
	CHM - Class Hierarchy of Method
	RM - Reuse Metric (of classes)
Chidamber 94	WMC - Weighted Methods per Class
	DIT - Depth of Inheritance Tree
	NOC - Number Of Children
	CBO - Coupling Between Object classes
	RFC - Response For a Class
	LCOM - Lack of Cohesion Of Methods

Coppick 92	SSM - Software Science Metrics (Halstead)
	MCC - McCabe's Cyclomatic Complexity metric
Laranjeira 90	Size - Size of Object-Oriented system
Lee 93	MC - Method Complexity
	CC - Class Complexity
	HC - Hierarchy Complexity of system
	PC - Program Complexity
Li 93	DAC - Data Abstraction Coupling
	(Number of abstract data types)
	NOM - Number Of local Methods
	MPC - Message Passing Coupling (number of send statements in a class)
	Size1 - number of semi-colons in a class
	Size2 - number of attributes + number of local methods
Moreau 90ab	SSM - Software Science Metrics (Halstead)
	MCC - McCabe's Cyclomatic Complexity metric
	IL - Inheritance Lattice (stated, but no measure indicated)
	GSDM - Graph of Source and Destination of Messages (no measure given)
Sharble 93	WAC - Weighted Attributes per Class
	NOT - Number Of Tramps (count of extraneous parameters)
	VOD - Violations Of the law of Demeter [Lieberherr 89]
Tegarden 92	SSM - Software Science Metrics (Halstead)
	MCC - McCabe's Cyclomatic Complexity metric
	LOC - Lines Of Code
Williams 93	COU - Count Of Uses
	CBC - Count of Base Classes
	CSC - Count of Standalone Classes
	CCR - Count of number of Contains Relationships

6. Annotated Bibliography

This bibliography enables the practitioner to scan the literature on software measures and develop a suite of metrics that apply in his or her environment. Section 6.1 contains most of the recent articles that present a software measure and specifically indicate how the measure is determined. Section 6.4 includes texts on general topics directly related to object-oriented software design and software measure. Section 6.2 lists the key seminal works of Halstead, McCabe, and Albrecht, since they provide a historical base for the discipline. In the annotations below, we have attempted to elucidate the key contributions of each article or text.

6.1. Articles Related to Object-Oriented Measures

Abreu, Fernando B. & Carapuça, Rogério. "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework." *Journal of Systems Software 26*, 1 (July 1994): 87-96.

The authors provide a taxonomy for metrics of object-oriented products and processes. This taxonomy, TAPROOT, deals with both product and process metrics plus some "hybrid" metrics that measure both. The author's taxonomy is based on a Cartesian product of the two vectors: (design, size, complexity, reuse, productivity, quality) and (method, class, system). This produces eighteen possible cells into which a metric can reside. Class and system quality metrics that the authors suggest are based on counts of observed defects, failures, and time between failures. TAPROOT is presented as a starting point from which further refinement and verification can follow.

Aksit, Mehmet & Bergmans, Lodewijk. "Obstacles in Object-Oriented Software Development," pp. 341-358. *Proceedings: OOPSLA Conference.* ACM Press, October 1992.

Based on the results of some pilot studies, the authors have formed their own viewpoint of object-oriented methods and have documented some obstacles. The authors state that each phase in object-oriented software development can be subdivided into three sub-components: preparatory work, structural relations, and object interactions. A short summary of state-of-the-art object-oriented methods follows the subdivision taxa.

Banker, Rajiv D.; Kauffman, Robert J.; & Kumar, Rachina. "An Empirical Test of Object-based Output Measurement Metrics in a CASE Environment." *Journal of Management Information Systems 8*, 3 (Winter 1991): 127-150.

This 23-page article begins by reporting studies that indicate the use of CASE without having measurement programs in place. The authors' main thrust is the issue of output measurement in a CASE environment.

Their comments on function points (FP) are

> FP components do not follow naturally from an object-based CASE environment.
>
> Application of FP to CASE-generated code is subjective and inconsistent.
>
> Albrecht's original FP weights need to be re-calibrated for CASE tools.
>
> The usual Technology-Complexity-Factor (TCP) adjustment for FP may need revised for CASE since TCP is based on 3GL development.

The authors propose a short-form variation of FP called Raw-Function-Counts and two new object-based output measures, Object-Counts and Object-Points. The authors statistically validate the various metrics to estimate effort, and their results are significant. These proposed measures worked well in the CASE environment created by the ICE software. The authors conclude, "Since objects were found to match project managers' mental model of the functionality of software developed with object-based CASE, information about objects would be useful to promote improved software development process control."

Byard, Cory. "Software beans: Class metrics and the mismeasure of software." *Journal of Object-Oriented Programming 7*, 5 (September 1994): 32-34.

This non-technical article discusses "why measure software," "class metrics," and "mismeasurement." The author comments, "class metrics do not measure complexity, do not measure the size of an application, and do not measure the quality of software." Class metrics "are indicators of programming style." The author concludes, "The key is not measurement, but process"; and, "developing new measures that analyze implementation vocabulary complexity, module cohesion and coupling, and development progress will help."

Chen, J-Y. & Lum, J-F. "A New Metric for Object-Oriented Design." *Information of Software Technology 35*, 4 (April 1993): 232-240.

The authors use Basili's Goal-Question-Metric model to develop metrics for complexity for object-oriented design. The authors propose eight metrics that are identifiable at the design stage:

1. operation complexity metric
2. operation argument complexity metric
3. attribute complexity metric
4. operation complexity metric
5. class coupling metric
6. cohesion metric
7. class hierarchy metric
8. reuse metric

Metrics 1 through 3 are subjective in nature; metrics 4 through 7 involve counts of features; and metric 8 is a boolean (0 or 1) indicator metric. To validate these metrics, the authors conduct an experiment involving six "experts" whose subjective class scores are regressed against the eight metrics. The resulting regression equation is used to score future object classes. The paper does not report the original data, the complete SAS output, or the criteria that the "experts" use to measure complexity.

Chidamber, Shyam R. & Kemerer, Chris F. "Towards a Metrics Suite For Object Oriented Design," pp. 197-211. *Proceedings: OOPSLA'91*. Phoenix, Arizona, October 6-11, 1991. New York, New York: ACM SIGPLAN Notices, 1991.

The authors propose six metrics that they evaluate relative to seven of Weyuker's properties. The authors' objective is to propose metrics that are not language specific. They introduce measures that capture some features such as coupling, cohesion, complexity, scope, and methods (defined as responses to possible messages).

Chidamber, Shyam & Kemerer, Chris F. "A Metrics Suite for Object-Oriented Design." *IEEE Transactions on Software Engineering 20*, 6 (June 1994): 476-493.

The authors use the theoretical base for ontological principles proposed by Bunge as a means of establishing a basis upon which to discuss the object-oriented metrics suite. Much of the material in the first four pages is the same as in their earlier paper in 1991. The authors define six metrics and evaluate them with respect to six of Weyuker's nine properties. They propose six metrics for object classes:

1. Weighted Methods per Class (WMC).
2. Depth of Inheritance Tree (DIT).
3. Number Of Children (NOC), number of immediate subclasses subordinate to a class in the hierarchy.
4. Coupling Between Object classes (CBO).
5. Response For a Class (RFC), cardinality of the set of all methods that can be invoked by some method in the class.
6. Lack of Cohesion Of Methods (LCOM), the number of method pairs whose similarity is zero minus the counts of the method pairs whose similarity is not zero.

These metrics are based on three assumptions: the inheritance tree is full, two classes can have a finite number of identical methods, and certain counts of

features are random variables that are identically and independently distributed.

Churcher, Neville & Sheppard, Martin J. "Towards a Conceptual Framework for Object-Oriented Software Metrics." *Software Engineering Notes 20*, 4 (April 1995): 69-75.

The authors caution that software measures for O-O systems present significantly greater challenges than their conventional counterparts. They propose a set of terms to serve as a basis for comparison of models of O-O systems. They emphasize the problems arising from different interpretations of coupling and uses. They summarize by stating "it seems premature to proceed with the speculative development of specific metrics due to the absence of a satisfactory framework fro their validation."

Coppick, Chris J. & Cheatham, Thomas J. "Software Metrics for Object-Oriented Systems," pp. 317-322. *Proceedings: ACM CSC '92 Conference.* Kansas City, Missouri, March 3-5, 1992. New York, New York: ACM Press, 1992.

The authors extend the Halstead metric and McCabe metric to object-oriented software design. The authors' examples are in LISP Flavors. An undefined tool (code not included) is applied to LISP source code, and the usual software science metrics are computed. The authors count the number of methods and observe that increased abstraction reduces programming effort. Nothing concrete is done with McCabe's metric.

Gowda, Raghava G. & Winslow, Leon E. "An Approach for Deriving Object-Oriented Metrics," pp. 897-904. *Proceedings: IEEE 1994 National Aerospace and Electronics Conference.* Dayton, Ohio, May 23-27, 1994. Los Alamitos, California: IEEE Computer Society Press, 1994.

The authors comment, "The object-oriented metrics proposed so far seem to concentrate on the design of a single class or the class structure and ignore the overall design of the system and program." They propose a classification scheme for object-oriented metrics with the five categories of system metrics, subsystem metrics, class metrics, object metrics, and reusability metrics. They discuss and contrast each of the methodologies of Rumbaugh and Wirfs-Brock. The authors claim to have a list of metrics that can be applied to some of the phases of each methodology. Although the authors actually list some features of the phase and methodology that can be measured, they do not indicate how to measure the feature.

Jones, Capers. *Programming Productivity.* New York, NY: McGraw-Hill, 1986.

The author summarizes the first 30 years of industrial and commercial programming. The first two chapters of this four-chapter book are about the

science of measurement and serve as an excellent introduction to the topic of measurement. In the third chapter, the author isolates 20 factors, supported by historical data, that have affected programming productivity.

1. The language used
2. Program size
3. Personnel experience
4. Requirements
5. Complexity of program & data
6. Use of structured methods
7. Program class
8. Program (application area)
9. Tools & environment
10. Enhancing existing systems
11. Maintenance
12. Reuse (modules & design)
13. Code generators
14. 4GLs
15. Separation of development locales
16. Defect detection & removal
17. Documentation
18. Prototyping
19. Project teams & organization
20. Morale & compensation of staff

Chapter four explores the intangible factors, which are not readily quantifiable, that affect productivity. These factors include size of staff and enterprise, stability during the project, training for staff and users, computing facilities, legal issues, project measurement mechanisms, outsourcing, project dynamics, and user participation among all of these. This is a good book for the beginning software engineer. Jones has a second edition of this work in publication.

Laranjeira, Luiz. "Software Size Estimation of Object-Oriented Systems." *IEEE Transactions on Software Engineering 16*, 5 (May 1990): 510-522.

The author presents a size estimation model that takes advantage of the characteristics of object-oriented systems and their specification. He also provides a confidence interval for the expected system size. COCOMO is applied in this setting to produce cost estimates.

Lee, Yen-Sung; Liang, Bin-Shiang; & Wang, Feng-Jian. "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow," pp. 302-310. *Proceedings: CompEuro.* Paris-Ivry, France, May 24-27, 1993. Los Alamitos, California: IEEE Computer Society Press, 1993.

The authors use Weyuker's nine properties as a basis of evaluation. They define four metrics: method complexity (MC), class complexity (CC), hierarchy complexity (HC), and program complexity (PC). These measures are based on various forms of the basic model:

size*(input coupling + output coupling)^2

None of the proposed metrics satisfy Weyuker's seventh property.

Li, Wei & Henry, Salley. "Maintenance Metrics for the Object Oriented Paradigm," pp. 52-60. *Proceedings: First International Software Metrics Symposium.* Baltimore, Maryland, May 21-22, 1993. Los Alamitos, California: IEEE Computer Society Press, 1993.

The authors state that metrics for the object-oriented paradigm have yet to be studied. Since terminology varies among object-oriented programming languages, the authors consider the basic components of the paradigm as objects, classes, attributes, inheritance, method, and message passing. They propose that each object-oriented basic concept implies a programming behavior. They include six metrics from Chidamber [Chidamber 91]:

Depth of Inheritance Tree (DIT)	Coupling Between Objects (CBO)
Number Of Children (NOC)	Response For Class (RFC)
Lack of Cohesion Of Class (LCOM)	Weighted Method per Class (WMC)

The authors construct a Classic-Ada metric analyzer to collect metrics from Classic-Ada design and source code. They define five additional metrics to complete the modeling:

Data Abstraction Coupling (DAC)	Number of Methods (NOM)
# of semicolons per class (Size1)	# of methods per # attributes (Size2)
Message Passing Coupling (MPC)	

A regression analysis is used with Change = number of lines changed in the artifact's history (classes) as the dependent variable. The authors' analysis of the results reveals that the metrics chosen (all 10) can predict the number of changes. There is no individual breakdown of which of these metrics is significant in the prediction.

Lieberherr, Karl J. & Holland, Ian M. "Assuring Good Style for Object-Oriented Programs." *IEEE Software 6*, 5 (September 1989): 38-48.

The authors put forward a simple law, the Law of Demeter, that encodes the ideas of encapsulation and modularity in an easy-to-follow form for object-oriented programmers. The law has two forms: class and object forms. The class form comes in two versions: minimization and strict versions.

Class minimization version - *Minimize the number of acquaintance classes over all methods.*

Class strict version - *All methods may have only preferred-supplier classes.*

Objects - *All methods may have only preferred-supplier objects.*

The motivation behind the Law of Demeter is to ensure that the software is as modular as possible. The law effectively reduces the occurrences of nested message sending and simplifies the methods.

Moreau, Dennis R. & Dominick, Wayne D. "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part I - The Methodology." *Journal of Object-Oriented Programming 3*, 1 (May/June 1990): 38-52.

The authors set forth three objectives for their research (paraphrased below):

1. Establish an evaluation methodology to measure impact of object-oriented design on the software development process.

2. Establish domain-specific problem decomposition and solution guidelines to support comparisons of object-oriented approaches.

3. Perform verification of object-oriented metrics.

The principles of the proposed method are based on user activities, are environment-independent, and are based on well-constructed experiments. The authors claim that the method is extensible, captures the structural object-oriented aspects of the software, and provides for the automatic capturing of the metrics-related data. The authors include Halstead's little n and big N metrics and McCabe's cyclomatic complexity metrics, along with two measures that are based on object-oriented features, a graph of the source and destination of all messages, and an inheritance lattice. This paper provides a clear overview of a method for measuring object-oriented software.

Moreau, Dennis R. & Dominick, Wayne D. "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part II - Test Case Application." *Journal of Object-Oriented Programming 3*, 3 (September/October 1990): 23-32.

In this companion article to their article above, Moreau and Dominick discuss a refinement of the objectives set forth previously into theoretical, methodological, developmental, and evaluative components. The methodology is applied in an interactive graphics application domain. The test case was completed in 11 phases:

1- Identify applications domain {interactive graphics editor}

2- Identify test development systems {C & C++}

3- Identify development paradigms {GKS for C & object-oriented for C++}

4- Identify metrics {those in Moreau [Moreau 1990a]}

5- Identify and classify development activities {three separate tasks}

6- Establish evaluative criteria {Basili's direct cost/quality criteria}

7- Develop environment independent experiments

8- Prepare environments {no functional differences}

9- Develop environment-specific experiments {8 subjects, 4 in each experimental group}

10- Perform experiments

11- Analyze results {non-parametric Wilcoxon statistics P=0.07}

The authors state, "This research has formally established the primary metric data definitions that completely characterize the unique aspects of object-oriented software systems, including the inheritance lattice and message graph."

Park, Robert E. *Software Size Measurement: A Framework for Counting Source Statements* (CMU/SEI-92-TR-20, ADA258304). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

This technical report presents guidelines for defining, recording, and reporting two frequently used measures of software size: lines of code and logical source statements. Park proposes a general framework for constructing size definitions and uses it to derive operational methods for reducing misunderstandings in measurement results.

Poulin, Jeffrey S. & Brown, David D. "Measurement-Driven Quality Improvement in the MVS/ESA Operating System," pp. 17-25. *Proceedings: Software Metrics Symposium.* London, U.K., October 24-26, 1994. Los Alamitos, California: IEEE Computer Society Press, 1994.

This paper describes experiences, quality initiatives, models, and metrics used to obtain quantifiable results in a large, complex software system. Although no object-oriented metrics were actually developed or computed, this paper shows that the introduction of object-oriented design and the construction of high quality reusable frameworks played a critical role in defect reduction.

Sharble, Robert C. & Cohen, Samuel S. "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods." *SIGSOFT Software Engineering Notes 18*, 4 (April 1993): 60-73.

This paper reports on research to compare the effectiveness of two methods for the development of object-oriented software. The two methods compared are responsibility-driven methods and data-driven methods. Each of the methods was used to develop a model of the same system. The authors use a suite of object-oriented metrics to collect measures of each model. The model developed with the responsibility-driven method was found to be less complex, to have less coupling between objects, and to have more cohesion within objects. The research produced three new metrics that can be useful for measuring object-oriented designs.

WAC - Weighted Attributes per Class.

NOT - Number of Tramps (number of extraneous parameters in signatures of methods of a class.

VOD - Violations of the Law of Demeter.

Symons, Charles. "Function Point Analysis: Difficulties and Improvements." *IEEE Transactions on Software Engineering 14*, 1 (January 1988): 2-11.

The author critically reviews Albrecht's function point analysis, proposes ways of overcoming the weaknesses identified, and validates by experimentation the proposed improvements. Some criticisms are that FPs underweight systems that are complex internally and FPs are not "summable." The author proposes the "Mark II" formula for information processing component size in unadjusted function points which is:

UFP = NI*WI + NE*WE + NO*WO

where

NI = number of input data elements

WI = weight of an input data type

NE = number of entity-type references

WE = weight of an entity-type reference

NO = number of output data element types

WO = weight of output data element type

Symons determines a set of weights from 12 systems and recalibrates these weights to match Albrecht's original UFP values for systems under 500 FPs. He concludes that

- Mark II involves an understanding of entity analysis, no conventions yet.
- Mark II has fewer variables to count, but more technical factors to consider.
- Albrecht's FP is not a technology-independent measure of system size and neither is Mark II, since a change in technology involves recalibrating.
- FP analysis works for business applications, but may not work well for scientific or technical applications.

Taenzer, David; Ganti, Murthy; & Podar, Sunil. "Object-Oriented Reuse: The Yo-yo Problem." *Journal of Object Oriented Programming.* (September/October 1989): 30-35.

The authors review two basic approaches to software reuse, construction, and inheritance, and present some basic problems and conflicts between encapsulation and inheritance. They discuss the basic styles for reuse of construction and subclassing. Based on their own experiences in reuse, the authors give examples of message control trees. This discussion leads to the definition of the "Yo-yo" problem, where resolutions of a message goes up and down the message tree.

Tegarden, David P.; Sheetz, Steven D.; & Monarchi, D.E. "Effectiveness of Traditional Metrics for Object-Oriented Systems," pp. 359-368. *Proceedings 25th Hawaii International Conference on System Sciences 4.* Kauai, Hawaii, January 7-10, 1992. Los Alamitos, California: IEEE Computer Society Press, 1991.

The authors begin by quoting Moreau: "traditional metrics are inappropriate for object-oriented systems for several reasons..." [Moreau 90]. This paper addresses two questions:

1. Can existing metrics developed for structured systems be used as effective measures of object-oriented systems? and

2. Can certain unique aspects of object-oriented systems be measured by traditional metrics?

In the background section, the authors create a small taxonomy of complexity based on Card and Conte [Card 90], [Conte 86]. They discuss the traditional SLOC, Halstead metrics, and the cyclomatic metric and these metric's potential use in the object-oriented setting. The authors conclude, "The use of the traditional metrics may be appropriate for the measurement of the complexity of object-oriented systems. Even though the order of magnitude of the traditional metrics may be suspect, the directionality seems to be correct."

Waguespack, Leslie J, Jr. & Badlani, Sunil. "Software Complexity Assessment: An Introduction and Annotated Bibliography." *Software Engineering Notes 12*, 4 (October 1987): 52-71.

The authors provide an introduction to software complexity and provide an exhaustive list of nineteen categories of complexity research. The works listed in the article cover the years 1974-1987, plus one entry from 1967. Some 500 works are listed in the form [Lastname##] where ## is the last two digits of the year, and 100 of these were selected for the annotated bibliography. The annotated bibliography contains the complete reference citation and the original abstract (or an excerpt from the work which portrays the author's intent) followed by the annotation.

Wang, A.S. & Dunsmore, H.E. "Early Software Size Estimation: A Critical Analysis of the Software Science Length Equation and a Data-Structure-Oriented Size Estimation Approach," pp. 211-222. *Proceedings: Third Symposium on Empirical Foundations of Information and Software Science.* Rosklide, Denmark, October 21-24, 1985. New York, New York: Plenum Publishing Co., 1985.

The authors address early size estimation by emphasizing the weaknesses of the current size estimation metrics in 1985. They conjecture that program size can be estimated as a function of some other measurable quantities related to the program. Empirically, data from Pascal programs suggest that the Halstead length equation is not suitable for predicting the size of large Pascal programs. The authors found that the count of the VAR (the number of unique variables) is an acceptable size estimation. Experimental results yield:

S = 102 + 5.31*VAR as an estimate with r=0.94 and mean

MRE = 0.30

Based on these results, early estimation of program size can be improved at the end of the design stage by using the VAR count. The authors caution that these are "lab" results, and software that was produced in the lab was not nearly as large as that produced in industry.

Weyuker, Elaine. "Evaluating Software Complexity Measures." *IEEE Transactions on Software Engineering 14*, 9 (September 1988): 1357-1365.

Weyuker establishes a standard for software measures in this seminal article. She states the conditions for a measure as follows:

"All the measures we consider depend only on the syntactic features of the program."

$P \equiv Q$ means that programs P and Q halt on the same input.

P;Q means that P is augmented by Q (a concatenation).

The measure of P is denoted by $|P|$.

The nine properties of measures:

1. $(\exists P)(\exists Q)(|P| \neq |Q|)$.
2. Let c be a number ≥ 0. Then there are finitely many programs of complexity c.
3. There are distinct programs P and Q such that $|P| = |Q|$.
4. $(\exists P)(\exists Q)(P \equiv Q$ and $|P| \neq |Q|)$.
5. $(\forall P)(\forall Q)(|P| \leq |P;Q|$ and $|Q| \leq |P;Q|)$.
6. $(\exists P)(\exists Q)(\exists R)(|P| = |Q|) \& (|P;R| \neq |Q;R|)$ and $(\exists P)(\exists Q)(\exists R)(|P| = |Q|) \& (|R;P| \neq |R;Q|)$.
7. There are program bodies P and Q such that Q is formed by permuting the order of the statements of P; and $|P| \neq |Q|$.
8. If P is a renaming of Q, then $|P| = |Q|$.
9. $(\exists P)(\exists Q)(|P| + |Q| < |P;Q|)$.

Williams, John D. "Metrics for Object Oriented Projects," pp. 13-18. *Proceedings: Object Expo Euro Conference.* London, U.K., July 12-16, 1993. New York, New York: ACM SIGS Publications, 1993.

The author poses the question, "Why metrics?" The answer, he says, is in both project management metrics and software development metrics. He proposes a "3db" curve for monitoring project progress. Neither the 3, the d, nor the b is defined. For software development, the author suggests using counts of "uses," counts of the number of base classes (classes that represent reused code), counts of stand-alone classes, and counts of the number of "contains" relationships in a

class. He comments, "depending on how deep a class is in the inheritance tree, it may have many 'hidden' members and methods."

6.2. Early Seminal (Much Quoted) Works of Significance to the Discipline

Albrecht, A J. "Measuring application development productivity," pp. 83-92. *Proceedings: IBM Applications Development Joint SHARE/GUIDE Symposium.* 1979.

This is the seminal work on function points. Albrecht's intent is to measure the amount of functionality in a software product based on either the coded product or a structured specification document.

Halstead, M. H. *Elements of Software Science.* North-Holland, NY: Elsevier Publishing, 1977.

This is the original early work on measuring coded software products based on lexical issues of the product, such as numbers of operators, operands, unique operators, and unique operands. The theory for both the length metric and the volume metric is based on principles of cognitive psychology and a subjectively determined constant called the Stroud Number.

McCabe, T.J. "A Complexity Measure." *IEEE Transactions on Software Engineering 2,* 4 (April 1976): 308-320.

McCabe's cyclomatic complexity metric is the first of the attempts at measuring complexity. The metric is based on the features of a directed graph representation of the software product.

6.3. Textbooks on Software Metrics

Card, David L. & Glass, Robert L. *Measuring Software Design Quality.* Englewood Cliffs, NJ: Prentice-Hall, 1990. ISBN 0-13-568593-1.

This short paperback text (104 pages plus appendices and references) is quite readable. The book proposes a small set of measures (referred to as "primitive design metrics") that are centered around design quality. The authors' intent is to provide the practitioner with criteria for improving software designs to promote productivity, quality, and maintainability. Most of the examples and data come from a structured design environment with FORTRAN as the language.

Conte, S.D.; Dunsmore, H.E.; & Shen, V.Y. *Software Engineering Metrics and Models.* Menlo Park, Calif.: Benjamin/Cummings, 1990.

This text presents the classical product measures, classical models of process, and the product and process measures currently available in the late 1980s. The authors include a chapter on experimental design and basic statistical inference. They present a set of model evaluation criteria that practitioners should find useful. They examine effort from two viewpoints, macro and micro environments, and include the classical studies that are associated with each of these environments. This text has worked well for me in a senior-level software measures class.

Fenton, Norman E. *Software Metrics, A Rigorous Approach.* London: Chapman & Hall, 1991. ISBN 0-412-40440-0.

This text is solid and well written. Chapter 1 motivates the discipline. Chapters 2 through 6 provide a coherent framework for the many diverse activities that comprise software metrics. Among these are measurement theory, design of experiments, and data collection. Chapters 8 through 13 cover process measures, product measures, and resource measures. The author has provided an extensive, partially annotated bibliography.

Zuse, Horst. *Software Complexity Measures and Methods.* Berlin: Walter de Gruyter & Co, 1990. ISBN 3-11-012226-X.

This is the most comprehensive coverage of software complexity measures available in 1990. The text covers the issue of "metric versus measure," discusses measurement, and discusses the various ways that data can be classified. The author includes at least ninety measures that have appeared in the literature (mostly European sources). The text is recommended as a reference for researchers and instructors.

6.4. Textbooks on the Object-Oriented Approach

Booch, Grady. *Object-Oriented Analysis and Design, Second Edition.* Redwood City, Calif.: Benjamin/Cummings, 1994. ISBN 0-8053-5340-2.

This is a solid text for learning the essence of the object-oriented approach. It covers the notation of the method, discusses analysis and design strategies, and contains an extensive bibliography. The text is a good reference book and a good text for an upper-level undergraduate class.

Coad, Peter & Yourdon, Edward. *Object-Oriented Analysis, Second Edition.* Englewood Cliffs, NJ: Yourdon Press, 1991. ISBN 0-1362-9981-4.

The authors cover object-oriented analysis in a straight-forward manner and introduce an object-oriented analysis (OOA) methodology consisting of five steps: identifying classes and objects, identifying structures, identifying subjects, defining attributes, and defining services. All of these items are combined into an "object diagram," which resembles a dataflow diagram combined with an entity-relationship diagram. The book's strength is the discussion of management issues that emerge from using object-oriented techniques.

Jacobson, I.; Christerson, M.; Jonsoon, P.; & Overgaard, G. *Object-Oriented Software Engineering.* Reading, Me.: Addison-Wesley, 1992. ISBN 0-2015-4435-0.

This text serves as a good introduction to the object-oriented technique. It presents object-oriented software engineering (OOSE) as a new methodology that emphasizes the interaction of the user with the system and emphasizes the problem domain. The text contains clear examples of the object-oriented approach at all levels of software development. Smooth reading of this text is impeded by frequent typographical errors.

Rumbaugh, J. et al. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991. ISBN 0-1362-9841-9.

This text is a solid coverage of the subject. The authors propose a complete methodology, the object modeling technique (OMT), which covers analysis, design, and implementation. The authors contrast their OMT with structured analysis and design and with Jackson's structured development method. For those of us who are familiar with the procedure-oriented techniques, the text provides a smooth transition to object-oriented techniques.

Wirfs-Brock, Rebecca; Wilkerson, B; & Weiner, L. *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice-Hall, 1991. ISBN 0-1362-9825-7.

The authors define the object-oriented approach and provide a complete coverage of object-oriented principles. They emphasize a responsibility-driven viewpoint of analysis and design that emphasizes clients and servers. They also suggest that quality of design can be measured by counts of the number of classes, the number of subsystems, the number of contracts per class, and the number of abstract classes. The diagrams are clear and reinforce the material.

6.5. Significant Articles on the Object-Oriented Approach

Coleman, D.; Dollin, C.; & Jeremaes, P. "Fusion: A Second Generation Object-Oriented Analysis and Design Method," pp. 189-193. *Proceedings: Object EXPO Europe Conference*. London, U.K., July 12-16, 1993. New York, New York: ACM SIGS Publishing, 1993.

The authors propose a method that incorporates the methods of Booch, Rumbaugh, and others to provide a direct route from requirements to implementation. This method, fusion, is a systematic method for use by medium- to large-size teams in the industrial environment and was developed at Hewlett-Packard Labs, Bristol, UK.

6.6. Texts on Mathematics and Statistics Relating to Measures

Mansfield, Maynard. *Introduction to Topology.* Princeton, NJ: Van Nostrand, 1963.

This is a classic text on topology. This small book (116 pages) covers the basics of point set topology at the undergraduate level and is a source of discussion for metrics and metric spaces.

Sachs, Lothar. *Applied Statistics: A Handbook of Techniques, Second Edition.* New York, NY: Springer-Verlag, 1984. ISBN 0-3871-6835-4.

This text is an excellent reference for statistical techniques and the concept of measuring phenomena so that they can be evaluated statistically. The text contains a wide range of tables of value to statisticians. It is also a good source of non-parametric statistical procedures.

7. Examples of Computation of Measures

In this section, we compute the measures that are representative of each taxon. We chose two examples that represent object-oriented design and implementation. The first is a computer performance simulation written in C++; the second is a car dashboard instrumentation written in Ada95.

7.1. Representative Measures for Each Taxon

As indicated in Section 5.1., many measures have been presented in the literature on object-oriented software product measures. We choose five measures, one representing each of our taxa, that we feel are representative of an entire suite of measures. These measures are listed in Table 7-1 below.

Table 7-1: Representative Measures

Taxon	Measure Chosen for Taxon	Description of Measure	Reference
Method	MCC	McCabe's Cyclomatic Complexity metric	[Tegarden 92] [McCabe 76]
Class	Size2	Total number of attributes and methods for a class	[Li 93]
Inheritance	DIT	The depth of the inheritance tree	[Chidamber 94]
Coupling and Uses	CCM	The summation of the number of accesses to other classes, the accesses by other classes, and the number of 'co-operating' classes.	[Chen 93]
System	SC1	The total number of edges in the hierarchy graph for the system.	[Abreu 93]

7.2. C++ Example (Computer Performance)

This example is based on the sample application program taken from the Johnsonbaugh and Kalin text on object-oriented programming in C++ [Johnsonbaugh 95]. The C++ implementation code for the methods is omitted, some documentation has been added, and the #include statements are not included. This example is not intended to be executable, but to emphasize the computation of the various measures that apply to an object-oriented software product.

This example, shown in Figure 7-1, is the design and top-level implementation of a software artifact to simulate the measurement of computer performance. The design consists of creating two base classes, BenchMark and Computer. Class BenchMark has JobA, JobB, and JobC as derived classes. Class Computer has DeskTop and Mainframe as derived classes, and DeskTop has WS (WorkStation) and PC (Personal Computer) as derived classes. A program, TestIt, simulates a computer "running" a benchmark and outputting the results of the test. The relationships between the base classes and the derived classes are evident in Figure 7-1.

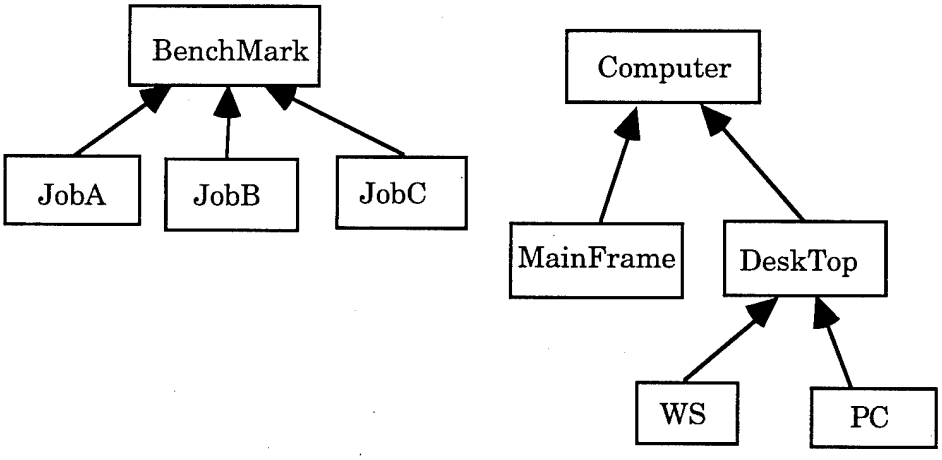


Figure 7-1: Class Hierarchy Chart

A partially coded implementation of the Computer Performance example follows:

```
const int MaxName = 100;
const float Tolerance = 0.01;
```

```
class Test;
class BenchMark  {
friend Test;
protected:
  //  Computer instructions are broken down into categories
  //    by percentage [ expressed as a decimal, 50% is 0.50 ]
  float  alP;      //  Arithmetic/logic instructions
  float  mP; //  Memory
  float  cP; //  Control instruction
  float  ioP;      //  Input/output instruction
  float  ic; //  Executed instruction count
  char  name[ MaxName + 1 ];
public;
  BenchMark()  //  base class constructor
  {
       init();
       strcopy( name, "?????" );
  }

  BenchMark( char* n )
  {
       init();
       //    includes if-then-else  stmt checking length of n
       //    McCabe's metric  (MCC)  is    2
  }

  void report()
  {
       //  cout statements to print values of variables
  }    //    MCC = 1

  int okay() // Checks to see if instruction percentages sum
             //    within tolerance to 1.0
  {
       return fabs(1.0 -(alP + mP + cP + ioP)) <= Tolerance;
  }    //    MCC = 2

  void init_error ()    //  Print error message when invoked
  {
       //     single cout statement
  }    //    MCC = 1
private:
  void init()      //    Initialize percentages to  0.0
  {
       alP = cP = mP = ioP = ic = 0.0;
  }
};     //    ===  end of class BenchMark  ===

class JobA :  public BenchMark {
//  This instantiation emphasizes arithmetic/logic and
//  control statements with moderate memory use and low I/O.
public:
  JobA() : BenchMark( "Job  A" )   //    JobA constructor
  {
       alP = 0.50;      cP = 0.20;
       mP = 0.20; ioP = 0.10;
       ic = ( float ) 4500301;
       if ( !okay() )  init_error;
```

```
  }
};      //   ===  end of class JobA  ===

class JobB :  public BenchMark {
//  This instantiation emphasizes arithmetic/logic and
//  control statements with light memory use and no I/O.
public:
  JobB() : BenchMark( "Job  B" )   //   JobB constructor
  {
        alP = 0.77;       cP = 0.166;
        mP = 0.064;       ioP = 0.0;
        ic = ( float ) 6700909;
        if ( !okay() )  init_error;
  }
};      //   ===  end of class JobB  ===
class JobC :  public BenchMark {
//  This instantiation emphasizes low arithmetic/logic and
//  control statements with heavy memory use & moderate I/O.
public:
  JobC() : BenchMark( "Job  C" )   //   JobC constructor
  {
        alP = 0.153;      cP = 0.0059;
        mP = 0.577;       ioP = 0.26;
        ic = ( float ) 10400500;
        if ( !okay() )  init_error;
  }
};      //   ===  end of class JobC  ===

class  Computer  {
friend  TestIt;
protected:
  //   cpi  =  cycles per instruction
  float  alcpi;     //  Arithmetic/logic cpi
  float  ccpi;      //  Control cpi
  float  mcpi;      //  Memory cpi
  float  iocpi;     //  Input/output cpi
  float  ct; //  Cycle time in nanoseconds
  char name [ MaxName + 1 ];
  float costU;      //   Upper bound of cost range in dollars
  float costL;      //   Lower bound of cost range in dollars

protected:
  Computer( float al,  float c,  float m,  float io,  float
t,  char* n,  float lbd,  float ubd )
  {
        alcpi = al;       ccpi = c;  iocpi = io;
        mcpi = m;  ct = t;
        if  ( strlen(n), MaxName ) strcpy( name, n );
        else  strncpy( name, n, MaxName );
        costU = ubd;      costL = lbd;
  }     //   MCC = 2
  void report ()
  {
        // cout statements to print cost range, time, and cpi
values
  }     //  MCC = 1
};      //  ===  end of class  Computer  ===
```

```
class Desktop:  public Computer  {
protected:
 Desktop(   float  al,       //  Arithmetic/logic
            float  c,        //  Control
            float  m,        //  Memory
            float  io,       //  Input/output
            float  t,        //  Cycle time in nanoseconds
            char*  n,        //  Name
            float  l,  //   Lower bound of cost range
            float  u ) //   Upper bound of cost range
       :  Computer( al, c, m, io, t, n, l, u )  {}
};     //   ===  end of class   Desktop  ===

class PC :  public  Desktop  {    //  Personal Computer
public:
 PC (  float  al = 1.8,      //  Arithmetic/logic
       float   c = 2.3,      //  Control
       float   m = 5.6,      //  Memory
       float  io = 9.2,      //  Input/output
       float   t = 230.0,    //  Cycle time in nanoseconds
       char*  n = "PC",      //  Name
       float  l = 800.0,     //   Lower bound of cost range
       float  u = 14500.0)   //   Upper bound of cost range
       :  Desktop ( al, c, m, io, t, n, l, u )  {}
};     //   ===  end of class   PC  ===

class WS :  public  Desktop  {    //  Workstation
public:
 WS (  float  al = 1.3,      //  Arithmetic/logic
       float   c = 1.7,      //  Control
       float   m = 2.1,      //  Memory
       float  io = 5.8,      //  Input/output
       float   t = 90.0,     //  Cycle time in nanoseconds
       char*  n = "WS",      //  Name
       float  l = 4500.0,    //   Lower bound of cost range
       float  u = 78900.0)   //   Upper bound of cost range
       :  Desktop ( al, c, m, io, t, n, l, u )  {}
};     //   ===  end of class   WS  ===

class Mainframe :  public  Computer  {  // Mainframe
public:
 Mainframe (     float  al = 1.2, //  Arithmetic/logic
       float   c = 1.5,      //  Control
       float   m = 3.6,      //  Memory
       float  io = 3.2,      //  Input/output
       float   t = 50.0,     //  Cycle time in nanoseconds
       char*   n = "$$$",    //  Name
       float   l = 310000.0, //  Lower bound of cost range
       float   u = 20000000.0) // Upper bound of cost range
       :  Computer( al, c, m, io, t, n, l, u )  {}
};
 //   ===  end of class   Mainframe  ===
```

```
class TestIt  {
  //    Computes the response time in nanoseconds of running
benchmark b
  //    on computer c,  where
  //          rt  =  response time,
  //          ct  =  clock cycle time,
  //          ic  =  instruction count, and
  //          cpi  =  clock cycles per instruction.
  //    The response time in nanoseconds is computed as
  //          rt  =  ic * cpi * ct .
  float rt;
  void results ( Computer c,  BenchMark b );
public:
  TestIt ( Computer c, BenchMark b );
};

int TestIt :: TestIT( Computer c, BenchMark b )
{
  float  al_rt,  c_rt,  m_rt,  io_rt;
  al_rt = b.alP * b.ic * c.alcpi * c.ct;
  c_rt = b.cP * b.ic * c.ccpi * c.ct;
  m_rt = b.mP * b.ic * c.mcpi * c.ct;
  io_rt = b.ioP * b.ic * c.iocpi * c.ct;
  rt = al_rt + c_rt + m_rt + io_rt;
  results (c, b);
}

void TestIt :: results ( Computer c, BenchMark b )
{
  //   cout  statements denoting computer and benchmark
  b.report();
  c.report();
}       //  MCC  =  1
//      ===  End of Class TestIt  ===
//      =====  End of Example  =====
```

7.2.1 Computation of Measures for C++ Example

The computer performance example in Figure 7-2 has two base classes, BenchMark and Computer.

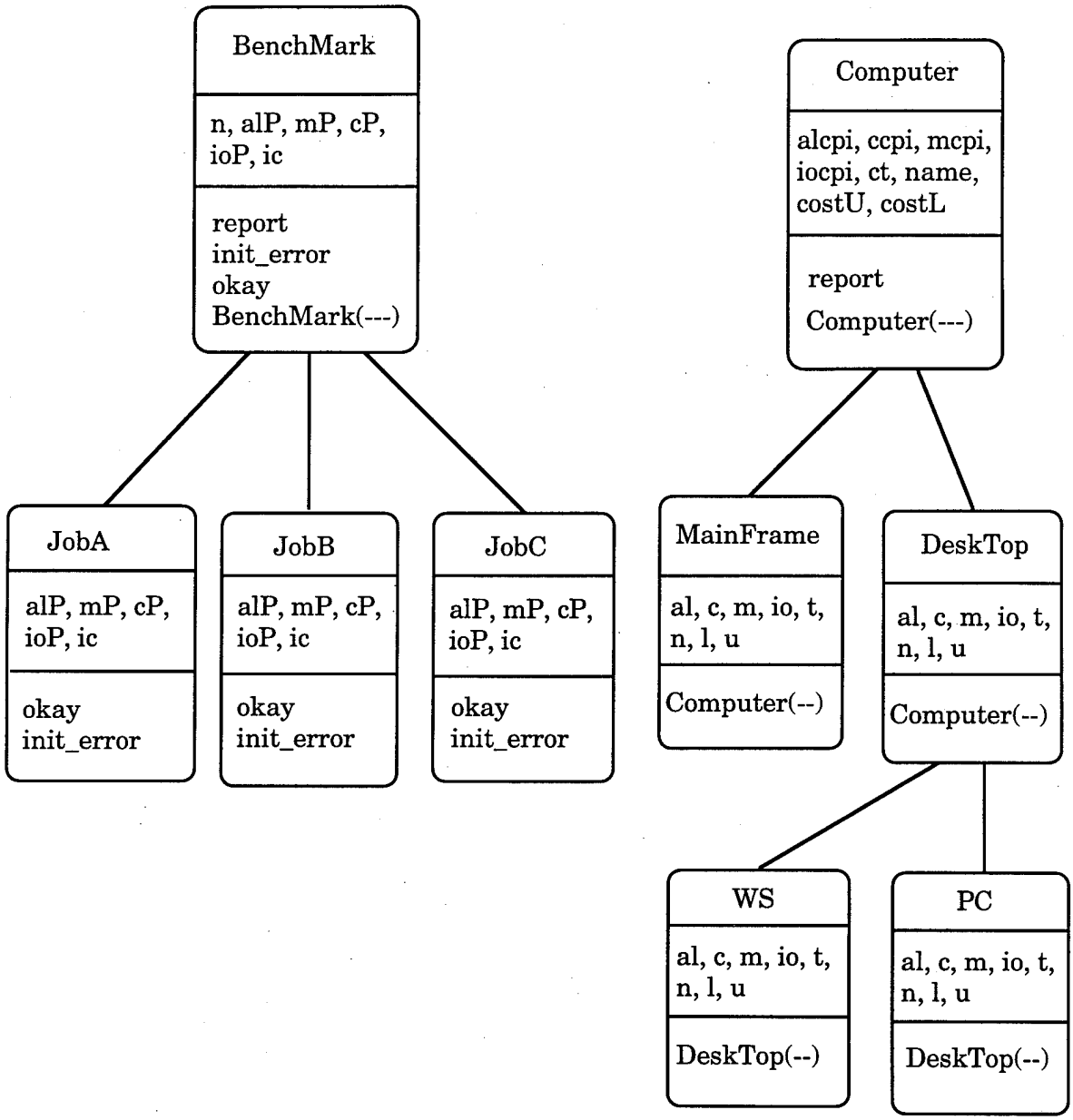


Figure 7-2: C++ Example Class Diagram

For the taxon *method,* McCabe's cyclomatic complexity (MCC) is calculated for each method in each of the classes. The base class BenchMark has four methods, which are inherited by three objects formed from this base class; so we need only consider these four methods for the base class BenchMark. Similarly, base class Computer has two methods, which are inherited by two objects formed from this base class; so we need only consider these two methods for the base class Computer. TestIt is a program that accesses the two classes to instantiate the objects. The results of the calculation of MCC are summarized below.

Class	Method	Measure
BenchMark	report	MCC = 1
	init_error	MCC = 1
	okay	MCC = 2
	Benchmark	MCC = 1
Computer	report	MCC = 1
	Computer	MCC = 2

For the taxon *class,* the measure Size2 proposed by Li and Henry [Li 93] is calculated for each class in the software system. Recall that Size2 is the total number of attributes and methods for each class. The results of the calculation of Size2 are summarized below.

Class	Measure
BenchMark	Size2 = 6+4 = 10
Computer	Size2 = 8+2 = 10

For the taxon *inheritance,* the measure is DIT (Depth of Inheritance Tree) proposed by Chidamber and Kemerer [Chidamber 94]. The results of the calculation of DIT are summarized below.

Class	Measure
BenchMark	DIT = 1
Computer	DIT = 2

For the taxon *coupling and uses,* the measure is CCM (total Count of the number of accesses to other Classes, accesses by other classes and the nuMber of cooperating classes) proposed by Chen [Chen 93]. The results of the calculation of CCM are summarized below.

JobA, JobB, & JobC access BenchMark	count is 3
DeskTop and Mainframe access Computer	count is 2
PC and WS access DeskTop	count is 2
Computer and BenchMark are accessed by TestIt	count is 2

CCM = 9

For the taxon *system,* the measure is SC1 (the total number of edges in the hierarchy graph for the system) proposed by Abreu and Carapuça [Abreu 94]. From Figure 7-2, the total number of edges in the hierarchy graph for the system is seven. (Simply count the arrow heads.)

SC1 = 7

7.3. Ada95 Example (Car Dashboard Instrumentation)

This example is based on the sample application program provided by the New York University GNU Ada Translator system (GNAT) [Schonberg 94]. The Ada95 implementation code for the methods is omitted, some documentation has been added, and the with and use statements are not included. This example is not intended to be executable, but to emphasize the computation of the various measures that apply to an object-oriented software product.

This example, the hierarchy of which is portrayed in Figure 7-3, is the design and top-level implementation of a software artifact to simulate some of the instruments on an automobile dashboard. The design consists of a base class, Instrument; and derived classes, Speedometer, Gauge, and Clock.

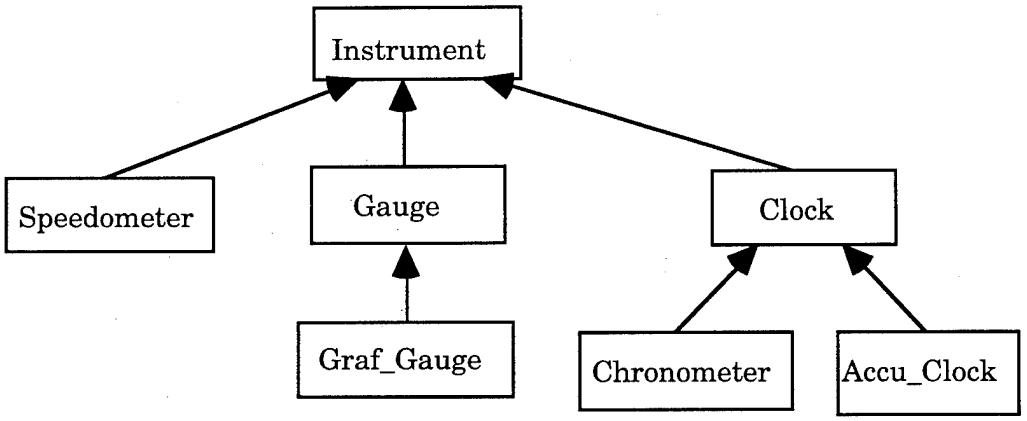


Figure 7-3: Ada Example Class Hierarchy Chart

A partially coded implementation of the Car Dashboard Instrumentation example follows:

```
package Instruments is
 ------------------------
 --     Root Type      ---
 ------------------------
 type Instrument is tagged record
      Name : String (1..14) := "              ";
 end record;
 procedure Set_Name (I: in out Instrument; S: String);
 procedure Display_Value (I: Instrument);

 ------------------------
 --   Speedometer      ---
 ------------------------
 subtype Speed is Integer range 0..85; --mph
 type Speedometer is new Instrument with record
      Value : Speed;
 end record;
 procedure Display_Value (S : Speedometer);

 ------------------------
 --     Gauges         ---
 ------------------------
 subtype Percent is Integer range 0..100;
 type Gauge is new Instrument with record;
      Value : Percent;
 end record;
 procedure Display_Value (G: Gauge);

 type Graf_Gauge is new Gauge with record
      Size : Integer  := 20;
      Full : Character  := '*';
      Empty: Character  := '.';
 end record;
 procedure Display_Value (G: Graf_Gauge);
 ------------------------
 --      Clocks        ---
 ------------------------
 subtype Sixty is Integer range 0..59;
 subtype Twenty_Four is Integer range 0..23;
 type Clock is new Instrument with record
      Seconds : Sixty  := 0;
      Minutes : Sixty  := 0;
      Hours   : Twenty_Four  := 0;
 end record;
```

```
  procedure Display_Value (C: Clock):
  procedure Init (C: in out Clock;
                  H: Twenty_Four := 0;
                  M, S: Sixty := 0);
  procedure Increment (C:in out Clock; Inc:Integer :=1);

  type Chronometer is new Clock with null record;
  procedure Display_Value (C: Chronometer);

  subtype Thousand is Integer range 0..999;
  type Accu_Clock is new Clock with record
       MSecs : Thousand  = 0;
  end record;
  procedure Display_Value (C: Acc_Clock);
end Instruments;  -- End Class Instruments --

-------------------------------------------
--  Program to test the Class Instrument --
-------------------------------------------
  procedure Test_Instruments is
  type ACC is access all Instrument'Class;
  package DashBoard is new Gen_List(Acc); use DashBoard;

  procedure Print_DashBoard (L: List) is
       L1 : List  := L;
       A  : Acc;
  begin
       while L1 /= Nil loop
         A := Element(L1);
         Display_Value(A.all);
         L1 := Tail(L1);
         end loop;
       New_Line;
  end Print_DashBoard;
```

```
--  >>>  Objects  <<<  --
Speed  : aliased Speedometer;
Fuel   : aliased Gauge;
Oil, Water : aliased Graf_Gauge;
Time   : aliased Clock;
Chrono : aliased Chronometer;
DB     : List  := Nil;
begin
      Set_Name (Speed, "Current speed");
      Set_Name (Fuel , "Fuel tank");
      Set_Name (Water, "Water ");
      Set_Name (Oil  , "Oil ");
      Set_Name (Time, "Current time");
      Set_Name (Chrono, "Chronometer");
      Speed.Value := 45;  --mph
      Fuel.Value  := 60;  --%
      Water.Value := 80;  --%
      Oil.Value   := 30;  --%
      Init (Time, 12, 15, 00);  --hrs, mins, sec
      Init (Chrono, 22, 12, 56);
      DB := Append (Speed'Access, Append (Fuel'Access,
            Append (Water'Access, Append (Oil'Access,
            Append (Time'Access, Chrono'Access)))));
      Print_DashBoard (DB);
end Test_Instruments;
```

7.3.1 Computation of Measures for Ada95 Example

This example, whose hierarchy chart is portrayed in Figure 7-4, has base class Instrument having three derived classes. The program Test_Instruments instantiates the objects to simulate the functions of the instrument panel.

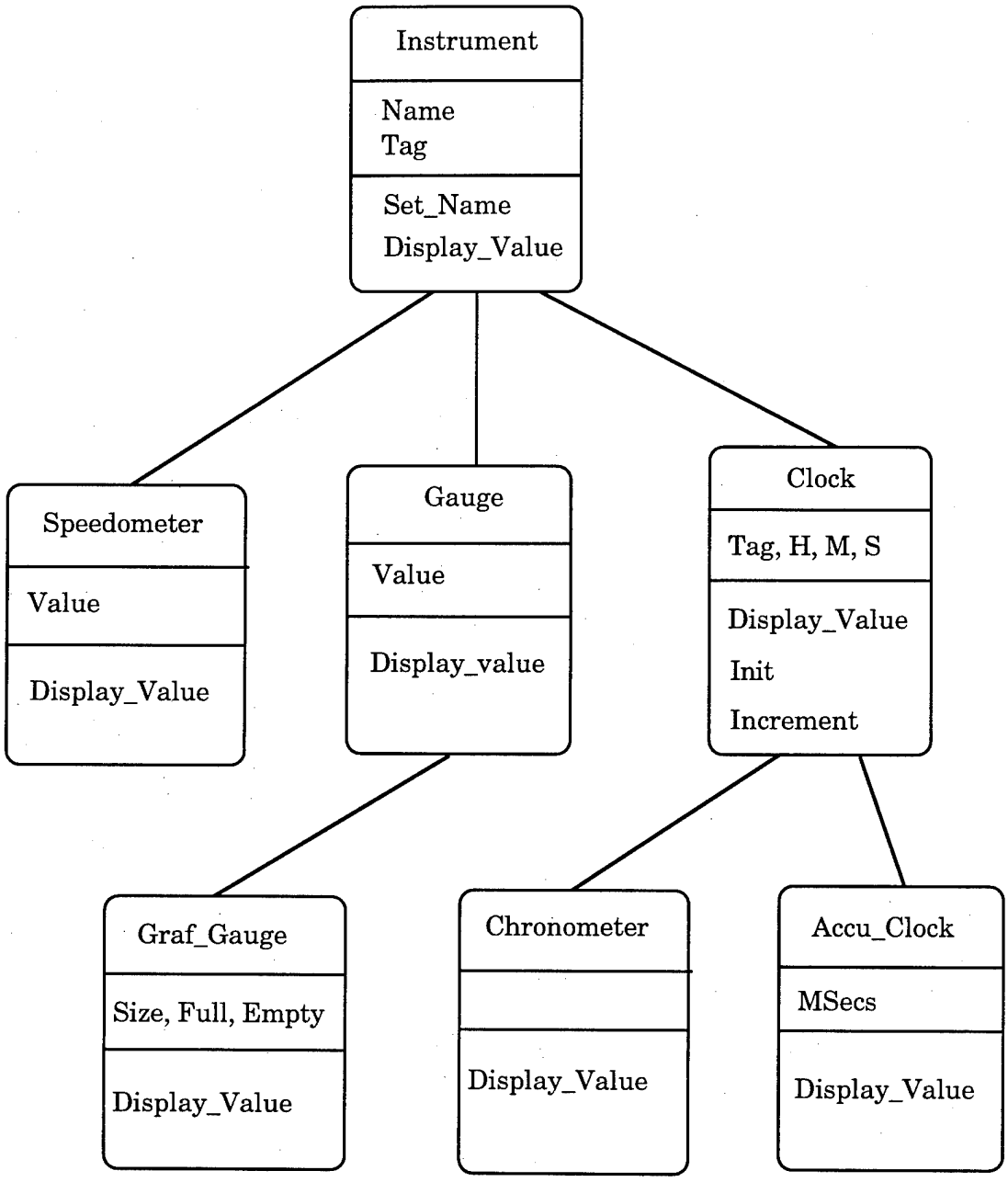


Figure 7-4: Ada95 Example Class Diagram

For the taxon *method,* McCabe's cyclomatic complexity (MCC) is calculated for each method in each of the classes. In Figure 7-4, we observe that the base class Instrument has two methods and the derived classes, Speedometer and Gauge, inherit these methods; so we need only consider the two methods. The derived class Gauge has a child class Graf_Gauge, which adds no new methods. Derived class Clock inherits Display_Value and adds two new methods, Init and Increment. Test_Instrument is a program which accesses the base classes to instantiate the objects. The results of the calculation of MCC are summarized below.

Class	Method	Measure
Instrument	Set_Name	MCC = 1
	Display_Value	MCC = 1
Clock	Init	MCC = 1
	Increment	MCC = 1

For the taxon *class,* the measure Size2 proposed by Li and Henry [Li 93] is calculated for each class in the software system. The results of the calculation are summarized below.

Class	Measure
Instrument	Size2 = 2+2 = 4
Speedometer	Size2 = 1+1 = 2
Gauge	Size2 = 1+1 = 2
Clock	Size2 = 3+3 = 6
Graf_Gauge	Size2 = 3+1 = 4
Chronometer	Size2 = 0+1 = 1
Accu_Clock	Size2 = 1+1 = 2

For the taxon *inheritance,* the measure is DIT (Depth of Inheritance Tree) proposed by Chidamber and Kemerer [Chidamber 94]. The results of the calculation are summarized below.

Class	Measure
Instrument	DIT = 2
Gauge	DIT = 1
Clock	DIT = 1

For the taxon *coupling and uses,* the measure is CCM (total Count of the number of accesses to other Classes, accesses by other classes and the nuMber of cooperating classes) proposed by Chen [Chen 93]. The results of the calculation of CCM are summarized below.

Speedometer, Gauge and Clock access Instrument	count is 3
Graf_Gauge accesses Gauge	count is 1
Chronometer and Accu_Clock access Clock	count is 2

CCM = 6

For the taxon *system,* the measure is SC1 (total number of edges in the hierarchy graph for the system), proposed by Abreu and Carapuça [Abreu 94]. From Figure 7-4, the total number of edges in the hierarchy graph for the system is six. (Simply count the arrow heads.)

SC1 = 6

8. Conclusions and Recommendations

The object-oriented paradigm is a model that represents the complexity inherent in large software systems. It measures complexity's many features (discussed by many of the authors in our bibliography) to enable the software engineer to monitor the software development process. It measures even those features that are in direct conflict with each other, such as execution time and memory space, efficiency and understandability, and coupling and cohesion.

There is, however, currently no single measure that captures all the features of an object-oriented software product. Based on this fact, a better approach to measuring object-oriented software products is to isolate the features of the product that are of concern to us and develop a suite of measures that measures these features. The two examples in Section 7 show that a suite of measures is easily calculated from the design documents. Most measures of methods also require actual computer code, or at least pseudocode. McCabe's measure, for example, was calculable from the incomplete code in the example, but Halstead's effort measure would have required executable code for its calculation.

The suite of measures we have used also has the property of consistency. For these measures, lower values were better than higher values. However, a few of these measures were 'weak' for the purpose used. McCabe's cyclomatic complexity measure was chosen as a member of the suite because it was calculatable from the incomplete code; however, the measure produced only values of one and two (indicating a lack of granularity). McCabe's measure is acceptable for identifying logic structures in programs but poor for measuring program size. Some authors have suggested augmenting the suite to include a size measure, a complexity measure, and an input/output count measure for the methods of a class, as opposed to collecting a single measure.

There are several key concepts captured in this report. First, once an analysis and design paradigm is chosen and the software products specified, a measurement plan should be put into action. The plan should identify

- the features of the product that require measurement,
- a process for selecting the appropriate measures,
- the data to be gathered, and
- consistent rules for calculating each measure.

Second, the object-oriented paradigm is continually evolving; the need for new measures to represent these added features is still there.

Acknowledgments

The authors would like to acknowledge Jorge Díaz-Herrera, Gary Ford, Linda Ibrahim, and Carol Sledge at the SEI and Gongzhu Hu at Central Michigan University for their valuable reviews of this paper; Linda Northrop for her advice on object-oriented methodologies; and Rachel Haas for editorial assistance.

References

[Abreu 94] Abreu, Fernando B. & Carapuça, Rogério. "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework." *Journal of Systems Software 26*, (1994): 87-96.

[Albrecht 79] Albrecht, A J. "Measuring application development productivity," pp. 83-92. *Proceedings: IBM Applications Development Joint SHARE / GUIDE Symposium.* 1979.

[Banker 91] Banker, Rajiv D.; Kauffman, Robert J.; & Kumar, Rachina. "An Empirical Test of Object-based Output Measurement Metrics in a CASE Environment." *Journal of Management Information Systems 8*, 3 (Winter 1991): 127-150.

[Booch 94] Booch, Grady. *Object-Oriented Analysis and Design, Second Edition.* Redwood City, Calif.: Benjamin/Cummings, 1994. ISBN 0-8053-5340-2.

[Carleton 92] Carleton, Anita D.; Park, Robert E.; Goethert, Wolfhart B; et al. *Software Measurement for DoD Systems: Recommendations for Initial Core Measures* (CMU/SEI-92-TR-19, ADA258304). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

[Chen 93] Chen, J-Y. & Lum, J-F. "A New Metric for Object-Oriented Design." *Information of Software Technology 35*, 4 (April 1993): 232-240.

[Chidamber 94] Chidamber, Shyam & Kemerer, Chris F. "A Metrics Suite for Object-Oriented Design." *IEEE Transactions on Software Engineering 20*, 6 (June 1994): 476-493.

[Coppick 92] Coppick, Chris J. & Cheatham, Thomas J. "Software Metrics for Object-Oriented Systems," pp. 317-322. *Proceedings: ACM CSC '92 Conference.* Kansas City, Missouri, March 3-5, 1992. New York, New York: ACM Press, 1992.

[DeMarco 82] DeMarco, Tom. *Controlling Software Projects Management, Measurement & Estimation.* Englewood Cliffs, NJ: Prentice-Hall. 1982.

[Ford 93] Ford, Gary. *Lecture Notes on Engineering Measurement for Software Engineers* (CMU/SEI-93-EM-9, ADA266959). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993.

[Halstead 77] Halstead, M. H. *Elements of Software Science.* North-Holland, NY: Elsevier Publishing, 1977.

[IEEE 94] IEEE Computer Society. "Metrics in Software." *Computer 27*, 9 (September 1994): 13-79.

[Johnsonbaugh 95] Johnsonbaugh, Richard & Kalin, Martin. *Object-Oriented Programming in C++.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

[Lee 93] Lee, Yen-Sung; Liang, Bin-Shiang; & Wang, Feng-Jian. "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow," pp. 302-310. *Proceedings: CompEuro.* Paris-Ivry, France, May 24-27, 1993. Los Alamitos, California: IEEE Computer Society Press, 1993.

[Li 93] Li, Wei & Henry, Salley. "Maintenance Metrics for the Object Oriented Paradigm," pp. 52-60. *Proceedings: First International Software Metrics Symposium.* Baltimore, Maryland, May 21-22, 1993. Los Alamitos, California: IEEE Computer Society Press, 1993.

[McCabe 76] McCabe, T.J. "A Complexity Measure." *IEEE Transactions on Software Engineering 2*, 4 (April 1976): 308-320.

[Moreau 90a] Moreau, Dennis R. & Dominick, Wayne D. "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part I - The Methodology." *Journal of Object-Oriented Programming 3*, 1 (May/June 1990): 38-52.

[Moreau 90b] Moreau, Dennis R. & Dominick, Wayne D. "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part II - Test Case Application." *Journal of Object-Oriented Programming, 3*, 3 (September/October 1990): 23-32.

[Schonberg 94] Schonberg, E. & Banner, B. "The GNAT Project: A GNU-Ada9X Compiler," pp. 48-57. *Proceedings: Tri-Ada 94.* Baltimore, Maryland, November 1994. New York, New York: ACM Press, 1994.

[Sharble 93] Sharble, Robert C. & Cohen, Samuel S. "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods." *SIGSOFT Software Engineering Notes 18*, 4 (April 1993): 60-73.

[Tegarden 92] Tegarden, David P.; Sheetz, Steven D.; & Monarchi, D.E. "Effectiveness of Traditional Metrics for Object-Oriented Systems," pp. 359-368. *Proceedings 25th Hawaii International Conference on System Sciences 4.* Kauai, Hawaii, January 7-10, 1992. Los Alamitos, California: IEEE Computer Society Press, January 1992.

[Tse 94] Tse, T. H. "Comparative Review of Object-Oriented Programming Texts." *Computing Reviews 35*, 4 (April 1994): 187-190.

[Weyuker 88] Weyuker, Elaine. "Evaluating Software Complexity Measures." *IEEE Transactions on Software Engineering 14*, 9 (September 1988): 1357-1365.

[Williams 93] Williams, John D. "Metrics for Object Oriented Projects," pp. 13-18. *Proceedings: Object Expo Euro Conference.* London, U.K., July 12-16, 1993. New York, New York: ACM SIGS Publications, 1993.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-95-TR-002	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-95-002
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-95-C0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63756E PROJECT NO.: N/A TASK NO: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Object-Oriented Software Measures	
12. PERSONAL AUTHOR(S) Clark Archer and Michael Stinson	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) April 1995 15. PAGE COUNT 55
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	annotated bibliography taxonomy measures object-oriented

19. ABSTRACT (continue on reverse if necessary and identify by block number)

This paper provides an overview of the merging of a paradigm and a process, the object-oriented paradigm and the software product measurement process. A taxonomy of object-oriented software measures is created, and existing object-oriented software measures are enumerated, evaluated, and placed in taxa. This report includes an extensive bibliography of the current object-oriented measures that apply to the design and implementation phases of a software project. Examples of computation of representative measures are included.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution	
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631	22c. OFFICE SYMBOL ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19